A General Logic Synthesis Framework for Memristor-based Logic Design

Zhenhua Zhu[1*], Mingyuan Ma[1*], Jialong Liu[1*], Liying Xu[2]
Xiaoming Chen[3], Yuchao Yang[2], Yu Wang[1], Huazhong Yang[1]
[1]Department of Electronic Engineering, BNRist, Tsinghua University, Beijing, China
[2]Department of Micro/nanoelectronics, Peking University, Beijing, China
[3]Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

***Abstract*—Memristor-based logic design gives an alternative solution to improve the energy efficiency of computing systems, benefiting from combining the memory with computing units. Inspired by this thought, previous work has demonstrated various memristor-based logic families with different attributes and computation patterns. Besides, some logic synthesis tools are designed for specific memristive logic implementations. However, the poor universality and the neglect of realistic constraints in memory largely restrict the utility of these logic synthesis tools. In this paper, we propose a general logic synthesis framework for memristor-based logic design, containing a universal abstract description method for memristive logic, a mapping rules generator, and a synthesis and mapping flow. The proposed logic synthesis framework is suitable for various types of existing memristor-based logic families and takes the memory status into consideration. It is also possible to handle future memristive devices and logic families by providing the universal abstraction interface. Furthermore, we also design a circuit-partitioning-based synthesis acceleration strategy to tackle with the long synthesis time problem. Experimental results show that, our framework can generate mapping results under the restriction of limited resource, while the existing synthesis tools may fail under the same restriction, and achieve comparable synthesis results with the same resource as the existing synthesis tools, which is enough for computation and storage. And the proposed acceleration scheme can achieve $\sim 1000\times$ speedup compared with the initial one.**

I. Introduction

In the post-Moore Era, the CMOS technology is approaching its physical device limits, which makes it hard to improve the energy efficiency of computing systems by further scaling down the feature size. Moreover, in the conventional von Neumann architecture, the huge data movements between the separated memory and processor consume the majority of the energy consumption of the entire system. For instance, a DRAM access costs three orders of magnitude more energy consumption than a singe arithmetic operation [1]. Thus, it is necessary to develop new computer architectures to break through the restriction of the energy wall in the von Neumann architecture.

An emerging nonvolatile memory technology, memristor, which is also known as resistive random-access memory, represents the stored data with different resistance values and has the capability of toggling the resistance by applying corresponding voltage/current. Recently, researchers utilize the resistance non-volatility and plasticity of memristors to design various memristor-based logic families (*i.e.*, memristive logic) in crossbar structures [2], which combine the memory with processing units and have the potential to overcome the memory wall problem.

*: All authors contributed equally to this work.

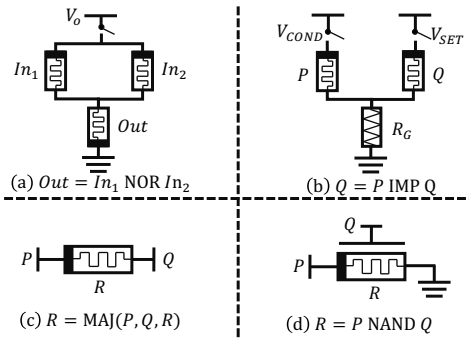

Fig. 1: Examples of memristive logic design: (a) `MAGIC` [3] (b) `IMPLY` [4] (c) `MAJ` [5] (d) Logic in 3-terminal devices [6]

In memristor-based logic design, the Boolean variables are represented by voltages and resistances, and the primitive logic gates are implemented by specific connection patterns of memristors. Figure 1 shows four typical memristor-based logic designs, which realize different basic Boolean functions by using different device characteristics, connection relations, and Boolean variable representations. More details will be discussed in Section II. Logic synthesis tools build a bridge between memristor-based primitive logic gates and large-scale logic circuits in practical applications. For existing logic synthesis tools, each tool is specifically designed for some logic family. For instance, `SIMPLE MAGIC`, proposed by [7], focuses on synthesizing logic functions and generating execution sequences onto `MAGIC` design; Refs. [8] and [9] design two different logic synthesis tools based on `IMPLY`; Ref. [5] presents a logic synthesis flow for `MAJ` by Majority-Inverter Graphs.

Despite of the effectiveness for specific memristor-based logic design, these existing logic synthesis tools are subject to two critical limitations in practice. First, the existing synthesis tools are poor in universality, *i.e.*, it is difficult to extend the compatibility of these synthesis tools with other memristive logic families. For example, in `MAGIC`, both inputs and outputs are represented by resistances, and `SIMPLE MAGIC` sets the gate parallel conditions by the position relationships between devices. However, in some logic families, the input variables are represented by voltages (*e.g.*, `MAJ`), and thus the parallel conditions of `SIMPLE MAGIC` are not suitable for these families. Besides, with the development of the device manufacturing technology, new device characteristics bring novel memristor-based logic designs. For example, Ref. [6] develops a type of 3-terminal memristors, which can implement new primitive logic gates compared with other memristive logic families. Existing logic synthesis tools are not applicable for

TABLE I: Overview of Existing Memristor-based Logic Synthesis Tools

Author	Memristive logic family	Synthesis Method	In-crossbar-logic	Memory usage considerations	Memory size constraints
[8]	IMPLY	SOP-based Synthesis	N	N	N
[9]	IMPLY	BDD-based Synthesis	N	N	N
[5]	MAJ	MIG-based Synthesis	N	N	N
[7]	MAGIC	ABC-based Synthesis	Y	N	N
[10]	FBLC	ABC-based Synthesis	Y	N	N
[11]	FBLC	ABC-based Synthesis	Y	N	N
This work	**All memristive logic**	**ABC-based Synthesis**	**Y**	**Y**	**Y**

such new devices and logic designs. Therefore, a general logic synthesis tool with high universality is needed to support the synthesis of existing or future memristor-based logic designs. Second, existing logic synthesis tools do not take the realistic memory constraints (*e.g.*, memory usage and memory size) into considerations, which makes the synthesis results can not be mapped to the practical memory. On one hand, some logic synthesis tools aim at generating the sequence of primitive logic operations and reducing the number of devices while not considering how to map the operation sequence into the memory (*i.e.*, memristor crossbars) [5], [8], [9]. On the other hand, other logic synthesis tools give the solution for memory mapping, but they assume the memory size is large enough and do not consider reusing the memristor storing intermediate data, causing waster of area and low memory utilization rate (*e.g.*, $< 60\%$). Besides, they also assume that the memory is unused before logic computations, which is not true in most cases [7], [10].

To tackle with the above limitations, we propose a general logic synthesis framework for memristor-based logic design, which is compatible with different memristive logic families and provides the mapping scheme with the consideration of realistic memory constraints. The main contributions include:

- We propose a general logic synthesis framework for different memristive logic families, which consists of a general abstract description method, a mapping rules generator, and a synthesis and mapping flow. It provides a general interface to describe any specific memristive logic design, generates synthesis constraints automatically, and outputs the detailed mapping results onto memristor crossbars. Experimental results show that, compared with existing studies, our framework can achieve widely generality, without any additional overhead on the synthesis results.
- The proposed logic synthesis framework takes realistic memory constraints into account. First, the memory usage before logic computations is taken into account to prevent the storage data from being corrupted by logic operations. Then, we utilize the overwriting of intermediate data in complex logic functions to perform computations within smaller hardware resources. Experiments show that our framework can generate mapping results with the realistic memory constraints, while existing work may fail under the same conditions.
- An acceleration strategy is presented to solve the long synthesis time problem caused by the huge exploration space. The acceleration strategy leverages the information of gate connections to divide the enormous design space into several local optimization sub-problems. Experimental results show that this acceleration strategy achieves more than $\sim 1000\times$ speedup compared with the initial one.

II. Preliminary

A. Memristor-based logic design

Memristor-based logic design performs logic operations in memristors with particular connection patterns, and expresses the Boolean variable with voltage or resistance. Ref. [2] divides typical memristive logic families into two categories, *i.e.* statefulness and non-statefulness, according to whether the Boolean variables are only represented by resistances and the computations are finished by changing the resistance states.

In statefulness logic designs, both inputs and outputs are represented by resistance. Two typical examples are MAGIC [3] and IMPLY [4], as shown in Figure 1 (a)(b). In these families, operating voltages are applied to the input devices. Then, the connection pattern and resistances of input devices (*i.e.*, input Boolean variable) will affect the voltage across the output device. Therefore, the output value is calculated by manipulating the resistance.

In non-statefulness logic designs, voltages are used as Boolean variables. We can further divide these families into two sub-categories depending on whether the output value is represented by resistance, *i.e.*, resistance outputs and voltage outputs. Two examples in the former sub-category are shown in Figure 1 (c)(d), these families use voltages to change the resistance value in the output device. In the latter sub-category, the output value is a voltage level. Two typical instances are MRL [12] and Pinatubo [13]: memristors are used as the voltage divider, and the output voltage is decided by input voltages and resistances. In such family, it is also necessary to convert voltage values into resistances for intermediate data storage in complex logic functions. Consequently, the non-statefulness logic designs can be regarded as one unified form: represent inputs and outputs with voltage (or together with resistance) and resistance, respectively.

B. Logic synthesis tools for memristive logic

The memristor-based logic design implements several completeness but simple primitive logic gates, which are far

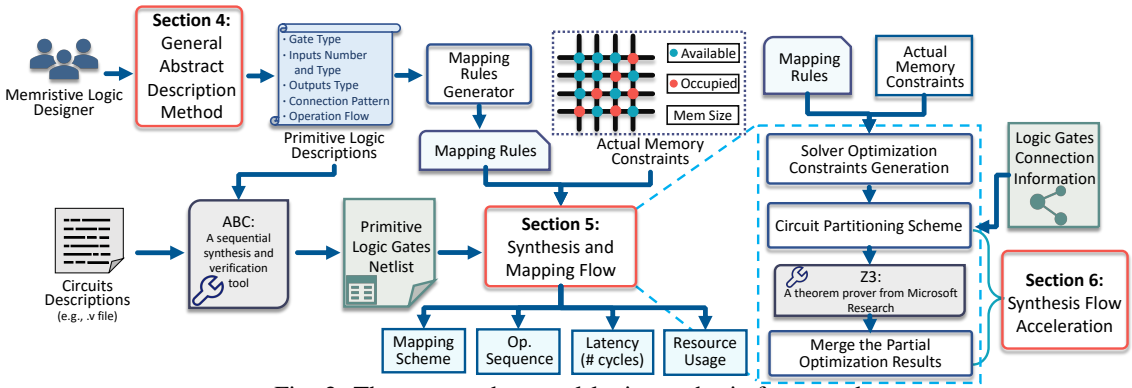

Fig. 2: The proposed general logic synthesis framework

away from complex logic functions used in actual application scenarios. Some logic synthesis tools are designed to reduce the gap between primitive gates and complex logic. Table I lists six typical logic synthesis tools. Five attributes are considered to evaluate these tools: 1. *Memristive logic family* refers to which memristor-based logic design can be synthesized by the tool; 2. *Synthesis method* indicates the algorithm for splitting the complex logic functions into primitive logic gates; 3. *In-crossbar-logic* indicates whether the synthesis tool considers mapping synthesis results onto crossbar structures; 4. *Memory usage considerations* mean whether the synthesis tool generates the synthesis results with the consideration of memory usage, such as stored data protection and erasing in memory; 5.*Memory size constraints* are used to indicate whether the logic synthesis tool takes into account the limitation of the actual memory array size.

From Table I, we see that existing logic synthesis tools are restricted by two factors:

(1) Each existing synthesis tool focuses on only one specific memristor-based logic design, making it hard to be used for synthesizing other memristive logic families with different primitive gates and Boolean variable representations;

(2) These logic synthesis tools do not take into account the realistic memory constraints in two aspects: 1. How to map the operation sequence (*e.g.*, logic gates netlist) onto crossbars that already store some protected data, and consider erasing and overwriting the intermediate data to reduce the usage of hardware resources; 2. Existing work performs synthesis flow under the assumption that the crossbar size is large enough, but how to generate synthesis and mapping strategy when the crossbar size is inadequate for the complex logic remains to be solved.

III. Overview of the General Logic Synthesis Framework

The overall logic synthesis framework is shown in Figure 2.

At the beginning of the entire logic synthesis flow, the proposed framework provides a general abstract description method for memristive logic designers. With the help of this method, the framework can synthesize and map the objective circuits with unified primitive logic descriptions, in spite of the specific logic design. Section IV gives more details about the general abstract description method.

After we get the primitive logic descriptions, the gate types information, together with circuits description files (*e.g.*, `Verilog` files), is sent to `ABC` tool, which is used for synthesis and verification of binary sequential logic circuits [14]. Then, the circuits are partitioned into multiple primitive logic gates and the gates netlist file is generated.

The core of the framework is the synthesis and mapping flow, whose details are demonstrated in Section V. The mapping flow is expressed as a gate location optimization problem, which contains four key points, *i.e.*, optimization tool, optimization constraints, optimization targets, and outputs. In our design, we use a Satisfiability Modulo Theories solver, *i.e.*, `Z3` Theorem Prover [15], to find the optimal gate locations. The constraints of this optimization problem consist of two parts: mapping rules and realistic memory constraints. The former one is produced by the mapping rules generator, and the latter one is given by the system profiling. After receiving the input gates netlist and constraints, `Z3` optimizes the computing latency (represented by the number of cycles) by trying different mapping schemes. After that, the mapping results (*i.e.*, gates locations and operations sequence) and system performance (*e.g.*, computing latency and resource usage) are presented.

Even if the synthesis and mapping flow can optimize the computing latency for any logic functions, the execution time can be extremely long (*e.g.*, days $\sim$ months), on account of the huge exploration space needs to be traversed by `Z3` solver. For dealing with this problem, we propose a synthesis flow acceleration strategy in Section VI. The acceleration strategy leverages the thought of divide and conquer, and partitions the large netlist file into several smaller-scale circuits considering the gates connection information. Afterwards, each partitioned part is optimized by `Z3` solver separately, and the local optimization results are merged into global output results.

IV. General Description of Memristive Logic

This section presents the general description of the memristor-based logic design. First of all, the theoretical analysis and the detailed description method are demonstrated in Section IV-A. Then, two typical case studies are provided in Section IV-B to verify the feasibility of the general description method.

A. General abstract description method

As demonstrated in Section II, the memristor-based logic designs are divided into two categories according to which physical quantities are used to represent Boolean variables.

For the statefulness logic gates, both inputs and outputs are resistance values. Different connection patterns, initial

TABLE II: Primitive Logic Gate Descriptions

Attribution	Parameter	Description
Primitive Logic Gate	TypeNum	# primitive gate types
	GatesType	The type of each primitive gate
For each primitive gate type		
Input Variable	InputNum	The fan-in of primitive gates
	InputMode	The representation of inputs: R (resistance) or V (voltage)
Intermediate Variable	IntraNum	The intermediates number
	IntraMode	The representation of intermediates
Output Variable	OutputNum	The fan-out of primitive gates
Connection Pattern	Constraint	Describe the location relationship between variables
Operation Flow	OpStepNum	The number of operation steps
	OpStep	Point out how to apply control signals in each step

resistances, and the operation steps determine the type of primitive logic gates in the statefulness logic families.

For the non-statefulness logic gates, voltages are also used to represent the Boolean variables. Besides, some memristive families use voltage levels as logic functions' outputs. However, considering the intermediate variables storage, the voltages need to be transformed to resistance values. Therefore, in our general description method, we assume all the memristive logic families utilize the resistance to represent the output value, while the original voltage outputs are described as intermediate variables. Based on this assumption, the functionality of the non-statefulness primitive logic gate is determined by variable representation methods, device characteristics, and voltages loading mode.

In summary, in order to describe an arbitrary memristor-based logic gate, these common characteristics are needed: variable representations, gate structures (connection patterns in each gate, device number, *etc.*), intial resistance, and operation steps. In our general description method, we propose a Boolean variable description triple to describe these characteristics as shown in Equation 1:

$$(B, r, c),\ s.t.\ B \in \{R, V\}, r \in \{WL\}, c \in \{BL\} \quad (1)$$

where B indicates the representation of a Boolean variable, *i.e.*, R means resistance and V means voltage, and (r, c) is the coordinate of the Boolean variable in memory crossbars, *i.e.*, r and c correspond to the position in Word-Line (WL) and Bit-Line (BL), respectively. For the record, for resistance values, the coordinate (r, c) stands for the memristor physical location in crossbars, while for the voltage value, (r, c) means the voltage loading position, $(-1, c)$ and $(r, -1)$ represent the BL voltage and WL voltage, respectively. Furthermore, on the basis of the above triple, different connection patterns can be expressed by a set of equations and inequalities about memristor coordinates. And an operation flow, *i.e.*, a sequence of control voltages and corresponding positions, describes initial resistances and the operation steps.

Based on the above idea, we design the general abstract description for memristive logic families. Table II lists all the attributions of primitive logic gates in this abstract description.

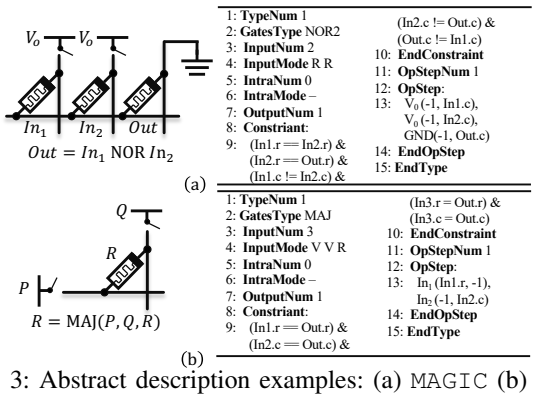

Fig. 3: Abstract description examples: (a) MAGIC (b) MAJ

B. Case study: Descriptions of MAGIC and MAJ

In this section, we choose two typical memristive logic families to demonstrate the feasibility of the proposed general abstract descriptions: MAGIC and MAJ, as shown in Figure 3. MAGIC is one statefulness logic family, whose primitive gate type is NOR, as shown in Figure 3(a). Figure 3(b) describes the MAJ design, which is a representative example of the non-statefulness logic design. MAJ uses original resistance (R) and the voltages at both ends of the memristor (P, Q) as inputs, and outputs the result into the memristor ($R = \mathrm{MAJ}(P, Q, R) = P \cdot R + \overline{Q} \cdot R + P \cdot \overline{Q}$). In these three examples, Line 9 (**Constraint**) gives the position constraints according to the connection patterns, and Line 13 (**OpStep**) provides the detailed operation flow.

V. Synthesis and Mapping Considerations

In our proposed logic synthesis framework, after the gate netlist file is generated by the ABC tool, the synthesis and mapping flow is modeled as a gate location optimization problem. Then we use the Z3 tool to search an optimal mapping scheme in the exploration space, for achieving the minimum latency. In this section, mapping constraints are discussed from the two aspects of parallel conditions (Section V-A) and realistic memory constraints (Section V-B). After that, we will introduce a gate operation model to describe operation stages and estimate the latency of logic gates in Section V-C.

A. Parallel conditions

Different from the CMOS-based logic gate design, memristive logic design performs logic operations in the crossbar structure, so the parallel conditions need to be considered carefully to avoid interference with other devices in the crossbar. [7] demonstrate the parallel manner of MAGIC, but the condition is quite different from other memristive logic families. In this section, we discuss the parallel conditions for statefulness and non-statefulness logic designs separately.

(1) For the statefulness logic design, the parallel conditions are determined by the connection pattern and operation steps. Specifically, two logic gates can be paralleled if and only if all the corresponding memristors in these gates are equivalent. For two memristors in logic gates, if the control voltages loaded to these memristors are identical (including voltage position, amplitude, and operation time), then we define these two memristors are equivalent.

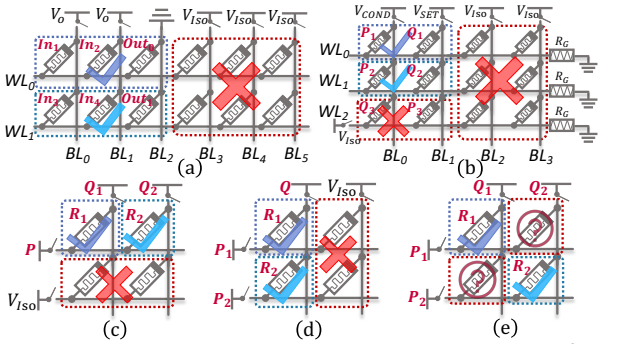

Fig. 4: Parallel conditions of logic design: (a) `MAGIC`: $Out = In_1$ NOR In_2; (b) `IMPLY`: $q = p$ IMP q; (c)∼(e) `MAJ`: $R =$ MAJ(P, Q, R). The gates marked with ticks/crosses can/cannot be executed in parallel respectively.

For example, in Figure 4(a), the **OpStep**s of the gates circled by the dark-blue box and the sky-blue box are the same, *i.e.*, $V_0(-1, In_1.c), V_0(-1, In_2.c), GND(-1, Out_0.c)$, so these two gates can compute concurrently (all the corresponding memristors are equivalent). Another example is shown in Figure 4(b), since `IMPLY` performs the operation $q = p$ IMP q, the two memristors (p and q) are unequivalent. So the gates located in WL_0 and WL_2 can not be executed in parallel. Besides, the left part and the right part of the crossbar can not be used at the same time (marked with red cross), for the reason that the control voltage positions are different (*i.e.*, $BL_0BL_1BL_2$ *v.s.* $BL_3BL_4BL_5$). Thus, the gates parallelism in the same row, which will make these two gates interfere with each other, is avoided.

(2) For the non-statefulness logic design, voltages are also used as input parameters. In the crossbar structure, the voltage signal is loaded to one WL/BL, resulting in all the memristors in the row/column are connected to it. Therefore, gates parallelism can only be performed in the following three cases, as shown in Figure 4(c)∼(e):

i. For $Gate_i$ and $Gate_j$, if $(i.In_1 = j.In_1)$, then $i.r = j.r$, $Gate_i$ and $Gate_j$ are located in the same row;

ii. For $Gate_i$ and $Gate_j$, if $(i.In_2 = j.In_2)$, then $i.c = j.c$, $Gate_i$ and $Gate_j$ are located in the same column;

iii. For $Gate_i$ and $Gate_j$, if $(i.In_1 \neq j.In_1)$ & $(i.In_2 \neq j.In_2)$, then $(i.r \neq j.r)$ & $(i.c \neq j.c)$, $Gate_i$ and $Gate_j$ are located in different rows and columns. At this time, the memristors with the coordinate of $(i.r, j.c)$ and $(j.r, i.c)$ will be written unexpectedly (marked with red question marks in Figure 4(e)). Therefore, in order to achieve a higher parallelism with more relaxed parallel conditions, only a small part of devices can be used for storage, causing a trade-off between resource utilization rate and parallelism.

B. Mapping strategy with realistic memory constraints

When the memristive logic goes to the realistic application, the mapping strategy should be adjusted according to the actual memory situation, which is not considered in existing work. In this section, we analyze the mapping strategy with realistic memory constraints, *i.e.*, memory usage and size limitation.

Memory usage: In our logic synthesis framework, the memristor usage is divided into two types: available and occupied. In the available memristors, we call the cells with the same resistance as the initial resistance as clean cells, and the rest as dirty cells. For the clean cells, no more other operations are needed before logic operations, and will not cause additional cycles. While for the dirty cell, even if it can be covered, one additional write is also needed for resetting to initial states. If the memristor is occupied, that is to say, the data stored in the cell can not be erased directly without backup. Then, if a logic gate needs to use the device at this location, two steps are performed: 1. Backup the storage data to an available memristor; 2. Overwrite the original occupied memristor. If the backup and the overwriting can be executed in parallel, then one cycle for overwriting and backup is required before performing the logic operation.

To represent the usage state of each memristor in the crossbar, we describe the memory usage with a state matrix, each value in the matrix represents the memristor usage state. The initial state matrix is regarded as an optimization constraint and fed into the framework.

Memory size limitation: Existing memristor logic synthesis tools first assume the crossbar is large enough, and then give the actual required crossbar size according to the synthesis results. So they can not complete logic synthesis and mapping under the given and limited crossbar size. In our design, the crossbar size is an optimization constraint parameter provided by users. When the crossbar size is enough, the synthesis and mapping flow is similar with existing work. But, if the crossbar size can not satisfy the existing mapping scheme, data transfer, intermediate overwriting, and backup of occupied cell are considered, as demonstrated before. Moreover, because the optimization target of the synthesis and mapping flow is the minimum latency, the optimization tool, *i.e.*, `Z3` tool, prefers to map the logic gate onto the clean cells, rather than the dirty cells and occupied cells with additional write cycles.

C. Gate operation model

For the purpose of describing the execution of logic gates, we design the gate operation model in the proposed logic synthesis framework. In the gate operation model, we divide the entire gate operation into three stages: **Input trans stage**, **Output clear stage**, and **Logic operations (ops) stage**. Compared with the existing synthesis and mapping tool [7], our gate operation model added two additional stages, *i.e.*, input trans and output clear, besides logical calculation, considering the following two factors:

(1) If the calculation cannot be completed in situ due to the memory usage and size limitation, we need data transfer and overwriting in order to perform the logic calculation.

(2) For some memristive families, the computation must be completed through overwriting units [4]. In order to complete the calculation and avoid data damage, additional data transfer and rewrite are needed.

The gate operation model is shown in Figure 5(a), and its details as shown below:

i. **Input trans stage:** First, we assume the beginning time of each logic gate is t_{start}. At t_{start}, the input trans time t_{trans} is determined by whether input data transfer and copy

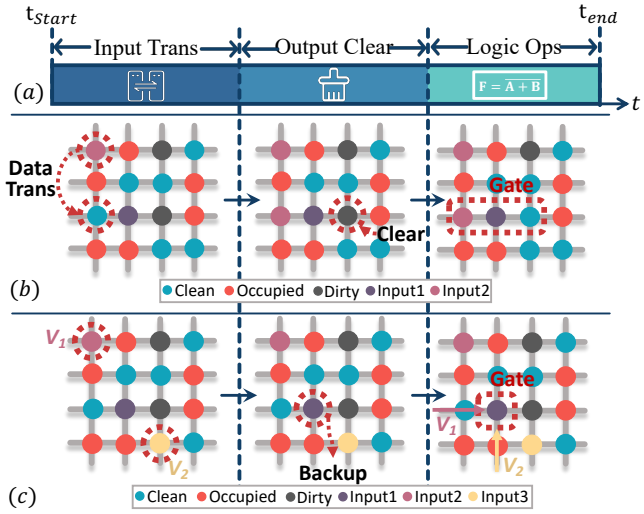

Fig. 5: (a) Gate operation model: the logic gate execution is composed of `Input Trans`, `Output Clear`, and `Logic Ops`; (b) Execution example of `MAGIC`, the steps of this gate are: input data trans, output location clear, and gate operation; (c) Execution example of `MAJ`, the steps of this gate are: read input data (R-to-V conversion), data backup, and gate operation

are required in this stage. Two judgements are considered here: 1. If the logic gate uses voltages as input variables, then the transformation from resistance to voltage is required, and the input trans time $t_{trans} = t_{R2V}$ (the transformation time from resistance to voltage, which is equivalent to the latency of a read operation); 2. For the resistance variable, if the original stored position does not match the logic gate position, the inputs need to be transferred to the new position with the input trans time $t_{trans} = t_{read} + t_{movements} + t_{write}$.

ii. **Output clear stage:** Similar with the previous stage, in the output stage, we take the output clear time t_{clear} into consideration when the cell at the output position is not clean. If the output memristor is dirty, we can clear it directly with $t_{clear} = t_{write}$. If it is occupied, in addition to the clear operation, backup operations are also required. Assume that the write operation in backup procedure can be paralleled with clear operation, then $t_{clear} = t_{write} + t_{read}$.

iii. **Logic ops stage:** This stage complete the logic operations with computing time t_{logic}, which is decided by the **OpStepNum** shown in the primitive logic gate descriptions.

Figure 5 (b)(c) give two logic gate execution examples according to the above three stages. Figure 5(b) demonstrates one `MAGIC` operation process, which consists of input data transfer, dirty output cell clear, and the statefulness logic operation. Figure 5(c) shows one example of `MAJ` execution, contains input variable conversion, occupied output cell backup and non-statefulness logic operation.

The completion time of the logic gate is shown in Equation 2:

$$t_{end} = t_{start} + \sum_{i=1}^{In} t_{trans,i} + \sum_{j=1}^{Out} t_{clear,j} + t_{logic} \quad (2)$$

In our latency estimation model, we think the transfer/clear process of different inputs/outputs must be executed serially, for the reason that simultaneously write two devices in the crossbar may cause mutual interference. Therefore, we accumulate the time of each transfer/clear operation to get a conservative but accurate result, as shown in Equation 2.

TABLE III: `MAGIC` synthesis time of examples with different number of gates

Gate Number	Synthesis Time	Crossbar Size
11	83s	8×8
13	230s	8×8
14	215s	8×8
18	1696s	8×8
21	2699s	8×8
35	$>$ 1 days	8×16
152	$>$ 1 week	24×24

D. Optimization problem

On the basis of the parallel conditions, realistic memory constraints, and the latency estimation model, the minimum latency mapping problem can be described as Equation 3:

$$\begin{aligned} &\min_{m} f(m), s.t., m \in \{Mapping\ Scheme\ Space\} \\ &f(m) = \max_{g} t_{end,g}, g \in \{Gates \text{ in mapping scheme } m\} \end{aligned} \quad (3)$$

In Equation 3, the exploration space (*i.e.*, the feasible mapping scheme space) is constructed under the parallel conditions and memory constraints. For each design point in the mapping scheme space, the circuit delay is defined as the largest completion time of all the gates. Each gate completion time goes from Equation 2, which is affected by the detailed mapping scheme. After the optimization problem model is established, we use `Z3` to traverse the exploration space and outputs the optimal design.

VI. Synthesis Framework Acceleration

As demonstrated in Section V, the synthesis and mapping problem is described as a memristor placement problem in crossbar. However, the exploration space is extremely huge because of the large number of gates and feasible gates locations. For example, Table III shows the synthesis time with different number of `MAGIC` gates, from which we see that the synthesis time increases dramatically with the increase of gate number. The long synthesis time is unacceptable when the gate number is large in the actual application scenarios, *e.g.*, 200 gates need several months for synthesizing.

In order to accelerate the synthesis and mapping flow, we design the circuit-partitioning-based synthesis framework acceleration strategy as shown in Algorithm 1, which reduces the synthesis and mapping time considering three key points:

(1) With the thought of divide and conquer, the original circuits netlist file is partitioned into several sub-blocks considering the gate labels: According to the maximum gate delay from the entire circuit input to the logic gate input, we label each logic gate with different levels (Line 1). By this label 2, different gates with the same label do not have sequential signal dependencies, and are more likely to achieve parallelism. Thus, we prefer to divide the logical gates with the same label into one circuit sub-block (Line 2∼Line 12); (2) Generally speaking, the smaller the level number, the more logic gates

Algorithm 1 The circuit-partitioned-based synthesis framework acceleration strategy

Input: The logic gates netlist file, mapping rules, crossbar size limitation S_{max}, and initial memory state matrix
Output: Mapping scheme, computing latency, resource usage.

1: For each gate g_i, calculate the label l_i;
2: Define the sub-block total number $N_{block} = 0$;
3: Set the gate number for each sub-block N_{gate};
4: **for** Each label l_i **do**
5: Partition $\sim N_{gate}$ un-partitioned gates with label l_i into one sub-block $block_j$;
6: $N_{block} + +$;
7: **for** Each gate g_{i+1} with label l_{i+1} **do**
8: **if** g_{i+1} is un-partitioned & all inputs come from $block_j$ **then**
9: Partition g_{i+1} in $block_j$;
10: **end if**
11: **end for**
12: **end for**
13: Take M_0 as a small part of initial memory state matrix
14: **for** Each sub-block $block_i$ **do**
15: Synthesis and map $block_i$ with M_i;
16: **if** $Size(M_i) < S_{max}$ **then**
17: M_{i+1} = Padding(M_i);
18: **else**
19: $M_{i+1} = M_i$
20: **end if**
21: Get the $mapping\ scheme$ and latency $latency_i$;
22: **end for**
23: Calculate $Latency$ with $\{latency_i\}$;
24: Calculate $Resource\ Usage$ with $M_{N_{block}}$
25: **return** $Latency$, $Mapping\ Scheme$, $Resource\ Usage$;

and outputs there are at this gate level. Therefore, for realizing the load balance, the circuit block in the lower level has fewer logic gates; (3) In order to get the equivalent mapping results compared with the original mapping scheme, we propose the "maintain-padding-transfer" method to track the memory status between synthesizing different sub-blocks. That is to say, after finishing the synthesis of a sub-block, the crossbar state matrix is maintained and padded to a larger crossbar state matrix. After that, the new state matrix is sent to next sub-block as the memory constraints for synthesizing (Line 14∼Line 22).

VII. Experimental results

In this section, we evaluate the proposed general memristive logic synthesis tool with benchmarks shown in [16]. Firstly, in Section VII-A, we use six different benchmarks to verify the universality of the proposed general logic synthesis framework. Secondly, the comparison with existing specific memristive logic synthesis tools and the mapping results under different crossbar size limitations are provided in Section VII-B. Thirdly, in order to show the mapping results with realistic memory constraints, a case study (*i.e.*, 1-bit full adder) is discussed in Section VII-C. Finally, Section VII-D gives the results about the proposed acceleration strategy.

TABLE IV: Logic Synthesis Results of Three Typical Memristive Logic Design (L means latency)

Benchmarks	MAGIC (8×8 Xbar)		IMPLY (4×8 Xbar)		MAJ	
	L	#Cell	L	#Cell	L	#Cell
C17	8	19	9	13	6	8
majority	9	19	11	10	8	9
clpl	14	29	12	14	11	15
t	8	16	9	11	5	9
newtag	10	25	10	15	9	11

TABLE V: Comparison With Existing Work: Synthesis and Mapping Results Under Different Crossbar Sizes (L means latency, '-' means no feasible solution found)

Benchmarks	Xbar Size	[7]		Ours	
		L	#Cell	L	#Cell
C17	8×8	10	18	8	19
	4×8	10	18	9	18
	4×4	-	-	12	16
majority	8×8	9	19	9	19
	4×8	9	19	9	19
	4×4	-	-	11	16
t	8×8	8	16	8	16
	4×8	8	16	8	16
	4×4	-	-	10	14

A. Universality verification

Table IV shows the logic synthesis results based on three typical memristor-based logic design, *i.e.*, `MAGIC`, `IMPLY`, and `MAJ`. In this results table, L means latency, which is defined as the number of the total clock cycles, $\#Cell$ is the number of the memristor actually used, $Xbar\ Size$ is the crossbar size.

B. Mapping results with different crossbar size constraints

Because the existing synthesis tools of `IMPLY` and `MAJ` do not consider mapping the logic gate onto crossbar structure [5], [8], [9], we only choose `SIMPLE MAGIC` [7] as a contrast object in this section. The comparison results are shown in Table V. From the results we see that, when the crossbar size is large enough, our proposed logic synthesis tool can achieve comparable results compared with existing work. When the crossbar size is limited, existing work is failed for the reason that it does not consider the overwriting of intermediate data and the resources are insufficient, but our tool can complete the computation by transferring and overwriting data. Besides, as the crossbar size decreases, the latency gets longer because more data movements and overwriting are needed.

C. Case study for mapping with realistic memory constraints

For demonstrating the effect of memory constraints on the mapping results, we choose 1-bit full adder as a case study, and the mapping results are shown in Figure 6. When the crossbar is large enough, the adder mapping result is shown in [7], which uses one 4×12 (the number of memristors actually used is 21) and ten cycles for computing. When the crossbar size is limited (*e.g.*, 4×6), then the intermediate data overwriting must be considered to make full use of the

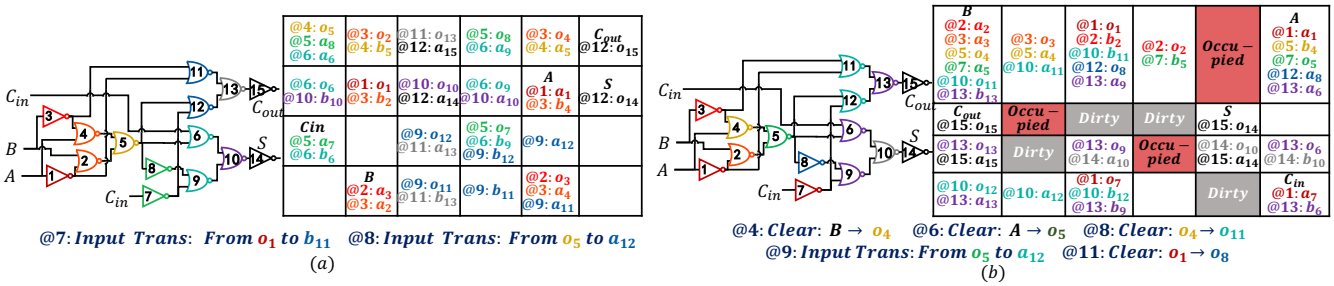

Fig. 6: A case study of 1-bit full adder on 4×6 crossbar: (a) mapping result in crossbar with limited size, (b) mapping results considering memory usage. For each logic gate, a and b represent the inputs and the output is denoted as o, $@t$ and the color indicate the gate execution time

TABLE VI: Acceleration Results (L means latency, the original synthesis time is longer than $2 \sim 7$ days)

Bench-marks	Before acceleration		After acceleration		
	L	Area	L	Area	Time (s)
5xp1	97	315	189	256	475.03
clip	136	444	237	324	954.32
misex1	45	294	94	144	821.34
parity	37	240	69	144	67.45
x2	36	168	91	144	273.21

resources. According to Figure 6(a), our proposed framework can complete the adder computation in 4×6 crossbar with only two additional cycles for data transfer, half of the resources are saved compared with the existing work. Besides, Figure 6(b) provides a mapping result considering realistic memory usage, *i.e.*, occupied cells and dirty cells. From the result we see that, except the occupied cell, more clear operations and data movements are required for computing in fewer devices.

D. Framework acceleration results

In order to test the acceleration results, we choose five different benchmarks, whose gates numbers are $68 \sim 152$. Because of the huge exploration space, the synthesis and mapping time is more than $2 \sim 7$ days, which is unacceptable in real case. To overcome this challenge, we propose the synthesis framework acceleration strategy, and the results are shown in Table VI. From the results we see that, compared with the original one, the speedup is about $1000\times$ with longer latency and smaller area. Because we use the partitioned strategy for acceleration, additional intermediate data storage and movements are required between different circuit sub-block, thus the acceleration strategy requires extra latency overhead. If we enlarge the gate number of each circuit sub-blocks and reduce the number of circuit sub-blocks, the latency results will get better, while the synthesis time will also increase dramatically. Thus there is a trade-off between the synthesis results and synthesis time.

VIII. Conclusion

In this paper, we propose a general logic synthesis framework, which applies to various existing memristive logic designs. It is also possible to handle future memristive devices and logic families by providing the universal abstraction interface. Experimental results show that, the framework performs comparably with state-of-the-art when the hardware resources are sufficient, and can generate feasible mapping results with realistic memory constraints, while existing work may fail. Besides, a circuit-partitioning acceleration scheme is presented to reduce the synthesis time and achieves $\sim 1000\times$ speedup compared with the original one.

IX. Acknowledgements

This work was supported by National Key Research and Development Program of China (No. 2017YFA0207600), National Natural Science Foundation of China (No. 61832007, 61622403, 61621091), Beijing National Research Center for Information Science and Technology (BNRist), and Beijing Innovation Center for Future Chips.

References

[1] S. Han *et al.*, "Eie: Efficient inference engine on compressed deep neural network," in *ISCA, 2016*, June 2016, pp. 243–254.
[2] J. Reuben *et al.*, "Memristive logic: A framework for evaluation and comparison," in *PATMOS, 2017*, Sep. 2017, pp. 1–8.
[3] S. Kvatinsky *et al.*, "Magic: Memristor-aided logic," *IEEE TCAS II*, vol. 61, no. 11, pp. 895–899, Nov. 2014.
[4] J. Borghetti *et al.*, "'memristive' switches enable 'stateful' logic operations via material implication," *Nature*, vol. 464, pp. 873–876, Apr. 2010.
[5] S. Shirinzadeh *et al.*, "Fast logic synthesis for rram-based in-memory computing using majority-inverter graphs," in *DATE, 2016*, March 2016, pp. 948–953.
[6] Y. Yang *et al.*, "Multifunctional nanoionic devices enabling simultaneous heterosynaptic plasticity and efficient in-memory boolean logic," *Advanced Electronic Materials*, vol. 3, no. 7, p. 1700032, July 2017.
[7] R. B. Hur *et al.*, "Simple magic: Synthesis and in-memory mapping of logic execution for memristor-aided logic," in *ICCAD, 2017*, Nov 2017, pp. 225–232.
[8] F. S. Marranghello *et al.*, "Sop based logic synthesis for memristive imply stateful logic," in *ICCD, 2015*, Oct 2015, pp. 228–235.
[9] S. Chakraborti *et al.*, "Bdd based synthesis of boolean functions using memristors," in *IDT, 2014*, Dec 2014, pp. 136–141.
[10] L. Xie, "Mosaic: an automated synthesis flow for boolean logic based on memristor crossbar," in *ASPDAC, 2019*. ACM, 2019, pp. 432–437.
[11] M. Traiola *et al.*, "Xbargen: A memristor based boolean logic synthesis tool," in *VLSI-SoC, 2016*, Sep. 2016, pp. 1–6.
[12] S. Kvatinsky *et al.*, "Mrl: Memristor ratioed logic," in *2012 13th International Workshop on Cellular Nanoscale Networks and their Applications*. IEEE, 2012, pp. 1–6.
[13] S. Li *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC, 2016*. ACM, 2016, p. 173.
[14] A. Mishchenko, "Abc: A system for sequential synthesis and verification," website, 2012, http://www.eecs.berkeley.edu/~alanmi/abc/.
[15] N. Bjorner, "The z3 theorem prover," website, 2012, https://github.com/Z3Prover/z3/graphs/contributors.
[16] S. Yang, *Logic synthesis and optimization benchmarks user guide: version 3.0*. Microelectronics Center of North Carolina (MCNC), 1991.